J Hardw Syst Secur (2017) 1:269–281
https://doi.org/10.1007/s41635-017-0015-0

A Methodology for the Characterisation of Leakages in Combinatorial Logic

Guido Bertoni[1] · **Marco Martinoli**[2] · **Maria Chiara Molteni**[1]

Received: 10 April 2017 / Accepted: 25 August 2017 / Published online: 30 November 2017

Abstract Glitches represent a great danger for hardware implementations of cryptographic schemes. Their intrinsic random nature makes them difficult to tackle and their occurrence threatens side-channel protections. Although countermeasures aiming at structurally solving the problem already exist, they usually require some effort to be applied or introduce non-negligible overhead in the design. Our work addresses the gap between such countermeasures and the naïve implementation of schemes being vulnerable in the presence of glitches. Our contribution is twofold: (1) we expand the mathematical framework proposed by Brzozowski and Ésik (FMSD 2003) by meaningfully adding the notion of information leakage, (2) thanks to which we define a formal methodology for the analysis of vulnerabilities in combinatorial circuits when glitches are taken into account.

Keywords Side-channel analysis · Hardware countermeasures · Glitches · Formal method

✉ Marco Martinoli
marco.martinoli@bristol.ac.uk

Guido Bertoni
g.bertoni@securitypattern.com

Maria Chiara Molteni
mariachiara.molteni@gmail.com

[1] Security Pattern, via Stassano 29, Brescia, Italy

[2] Department of Computer Science, University of Bristol, Bristol, UK

1 Introduction

Side-channel attacks were first introduced by Kocher et al. [10] as a way to attack implementations of cryptosystems. They exploit the relation between data being processed and several physical emanations, for instance time taken or power consumed to perform computations [11]. Since its first appearance, side-channel analysis has grown quickly with newly developed attacks as well as countermeasures, which try to prevent any sensitive information from being leaked. For instance, sharing schemes randomise intermediate values in such a way that the leaked information no longer depends on any sensitive data [13]. However, the efficiency of countermeasures is deeply linked to physical characteristics of the device on which they are implemented: in 2005, Mangard et al. [14] predicted the criticality of glitches for hardware implementations, which was then demonstrated in the same year [15]. They showed how the propagation of signals in combinatorial logic implementing an apparently secured SBox might result in critical leakages, leading to an ineffective protection.

Overall, there is a gap in the capabilities of quantifying the criticality of glitches in a hardware implementation. This gap is not trivial to close, as glitches in combinatorial logic are functions of the final layout of the circuit and the environmental conditions, and might change during the life of the device. In practice, two equal devices might exhibit a different behaviour in terms of glitches.

Our aim is to provide a formal framework for evaluating the presence of glitches under worst-case conditions without the need of detailed characterisation of the combinatorial logic, i.e. remaining at gate-level description. In order to achieve this result, we start from the mathematical structure

created by Brzozowski and Ésik [7], which simulates the propagation of electric signals inside a circuit, and we build a method to relate a modelled power consumption with the sensitive variables that have caused it. Our analysis is done in a worst-case scenario where all possible glitches are taken into account as to achieve the maximum possible generality. Our main result is an assessing tool which is able to formally describe what kind of information could be leaked and to give an heuristic estimate about the security of sharing schemes implemented in hardware.

Related Work To solve the problem of glitches, Nikova et al. [18, 19] suggested the use of threshold implementations, which allow to tackle glitches at root by developing maps that do not handle all the shares in the same combinatorial circuit. Such maps obviously come at the cost of a significant overhead compared to the unprotected version. Implementations and practical discussions can be found in the work of Moradi et al. [17] and of Bilgin et al. [6]. As for higher-order security, the issue of glitches has been faced with a generalisation of threshold implementations [5, 23], and independently by Prouff and Roche [20]. Specifically on the effects of glitches on the AES SBox, Mangard and Schramm [16] have reported a deep and complete analysis. From a design perspective, instead, some tools that attempt to identify leakages in masked circuits, also caused by glitches, already exist. Reparaz [22] described a methodology based on t test which, although similar in the goal, differs from ours in that it inherits the heuristic nature of the exploited statistical tools. Moreover, we do not require collection nor simulation of power traces. Also, the idea proposed by Leiserson et al. [12] is based on a heuristic method that allows the analysis of values flowing in more parts of a circuit through the so-called activity images. The main focus of their technique is on circuit modifications to thwart glitches' threat, while we propose an evaluation of already existing circuits without altering their structure. Finally, our approach is similar to the work of Tiwari et al. [24] but they only focus on how untrusted inputs propagate through a circuit to the output, while we mainly care of leaked intermediates. For this reason an important role in our work is played by input transitions, while their focus is more on fixed inputs.

Organisation of the Paper Section 2 provides the abstract framework underlying our tool, with a particular emphasis on how circuits, signals propagating inside them, power consumption and adversaries are modelled. Section 3 describes parts of the work of Brzozowski and Ésik [7] which are also used by our construction. In Section 4, we present our main contribution: we expand the functionalities of the previously discussed mathematical model with the notion of leakage and we show how such an improved framework can be used to analyse cryptographic circuits. The approach taken in this work is heuristic-oriented, for the sake of focusing on practical experiments of the model. For a more detailed and rigorous description of the latter two sections, we refer the reader to our previous work [4]. In Section 5, we test our tool and the underlying model with the sponge function KECCAK. We discuss the soundness of our approach and several practical aspects in Section 6, and we conclude our work in Section 7.

2 Preliminaries

Our work targets hardware implementations of cryptographic schemes. Since the meaning of such can be quite broad, the present section aims at specifying our environment, as well as at setting the notation we adopt. In fact, our mathematical model applies only to an abstraction of real-world circuits: we just refer to logic netlists; hence, circuits formed only of logic gates and connections among them. Our tool therefore achieves a good level of generality, since it does not require any knowledge of implementation details apart from the circuit scheme itself, which means that it is general enough to include all the previously mentioned source of glitches (final layout, environmental conditions...). In particular, we focus on asynchronous feedback-free circuits. We claim this is not too restrictive, because of the following argument. Circuits can be divided into two parts: the combinatorial logic and the state storing part. The combinatorial logic is indeed asynchronous; it is the part in charge of implementing the logic functionality and where glitches might propagate. The state storing part, implemented via registers or memory cells, is clocked and provides the synchronisation between different sections of the circuits. Since we apply our model to logic circuits performing sensitive computations, the most natural choice is to focus on the asynchronous part only. We do not consider the presence of feedbacks in the combinatorial part for the sake of simplicity and because they are not a common construction in this field anyway.

We adopt a high-level abstraction of signals. Since we are only interested in the Boolean value they represent, it is convenient to think of them as square waveforms which can assume the values 0 or 1. To push the abstraction further, we define the following mathematical object.

Definition 1 A *transient* is a bit-string with no repetitions. More formally, a bit-string $t = a_1 \dots a_x \in \mathbb{Z}_2^x$ is a transient if $a_i \neq a_{i+1}$ for all $1 \leq i \leq x - 1$. Notice that bit concatenation is denoted by simply writing one bit after the other. Moreover, we denote by T the set of all possible finite-length transients.

Informally, transients can only be of the form $1010\ldots$ or $0101\ldots$ for an arbitrary finite length $x \geq 1$ (note that bits 0 and 1 can be considered as transients when $x = 1$). The rationale behind transients is the following. Contracting bit strings is equivalent to neglecting time periods during which a signal assumes constant values 1 or 0. This results in transients being exclusively designed to represent which changes occur, but not when the order of switches can then be freely tuned, in such a way that the worst glitch behaviour is always shown at the output of a gate. That is to say if two transients modelling two changing signals are given as inputs to a gate, then the output will be a transient modelling the signal showing the highest possible number of changes. Section 3 specifies how to combine transients so to emulate gates' logic and to achieve such a functionality.

Further Notation We denote the power set (i.e. the set of all subsets) of a set S by $\mathcal{P}(S)$. Vectors are denoted by underlined letters while boldface is reserved for signals seen as transients (cf. Definition 1 and Example 1).

2.1 Power Consumption Model

If we consider global synchronous circuits, the power consumption can be divided in three components: the static leakage, the switching of registers and the switching of combinatorial logic. The static leakage is the amount of power needed by the circuit to maintain the current state when no switch is present. The switching of registers is the consumption taken by the circuit for updating the state and is easily approximated by the Hamming distance of the state in two consecutive clock cycles. The value of the registers can be easily protected by masking schemes. The last contribution is the most interesting for us and is related to the consumption of the combinatorial logic. From a temporal point of view, the switching of registers usually happens at the rising edge of the clock cycle while the static leakage happens in its last part. By contrast, the consumption of combinatorial logic spans, in most cases, the entire duration of the clock cycle [21].

Hamming Distance Model Consistently with the choice of addressing only the asynchronous part of a circuit, our power consumption model includes only the contribution of the combinatorial logic. As mentioned above, such part of a circuit is in charge for actually implementing the functionality of the circuit and it is the one where all the dynamic changes in values carried by wires happen. Moreover, when dealing with glitches, the main focus should be on how many times and in which moments a signal change in order to even recognise it as a glitch in the first place. For these reasons, a natural choice for our setting is the Hamming distance model, in which changes represent the most important metric. When modelling a gate's power consumption, it is therefore appropriate to consider the signal it outputs or, equivalently, the corresponding bit string. If the output signal changes, equivalently the corresponding output bit string switches, the gate consumes. In these terms, the Hamming distance model we assume in the present work is described by the following three assumptions:

1. A gate consumes power if and only if its output bit-string switches.
2. A zero-to-one switch consumes the same amount of power as a one-to-zero switch.

As already stated, we neglect static leakage by means of the first assumption. The second assumption is made for the sake of simplicity and it can be dropped in favour of a more realistic model built on top of a specific technology library. Nevertheless, such assumption is often use in the literature.

The above two statements merely refer to what is considered to be the power consumption from a mathematical point of view. They summarise what in literature appears as Hamming distance model. For our purposes, however, a further assumption is needed to relate how such consumption affects what we will define as leakage.

3. Every time some power is consumed, an attacker can measure and exploit it. Hence, we assume that a potential leakage exists as long as a switch occurs.

We will discuss in Section 2.2 more details on what type leakage an adversary can retrieve, and in Section 4 how we model leakage. In practice, the third assumption ensures the highest possible generality: we consider as leaked any variable that has a chance to be leaked.

2.2 d-Probing Model

Designing scheme being provably side-channel resistant comes with the intrinsic problem of mathematically formalising the environment in which a side-channel attack usually takes place. Many models have been proposed, each capturing certain aspects of practical attacks: e.g. whether the adversary has the ability to inject faults, to learn only a small amount of information being processed or to learn a more complete but noisy view on the internal state. In this respect, one of the most influential and seminal work was done by Ishai, Sahai and Wagner [9]. Among other noticeable achievements, they presented a mathematical framework in which it is possible to prove cryptosystems secure against adversaries who can probe a certain set of d wires in the implementation and learn the values they carry. The proof of security is based on the argument that, up to d probes, the view of the adversary is independent of any sensitive variable.

There are several reasons why the d-probing model has been so widely adopted. First of all, Ishai, Sahai and Wagner [9] built a generic compiler that can turn any cryptographic algorithm in a version secure in the d-probing model. The protection obviously comes with an overhead which is at least quadratic in the number of probes the adversary is allowed to use. Nevertheless, such an overhead is unavoidable if one aims for provable security. A further reason why to prefer the d-probing model over more sophisticated ones is its good adherence with what happens in practice: if on one hand it might seem unreasonable that a real adversary learns the exact value on certain wires, Duc et al. [8] have shown this is equivalent to security in the noisy leakage model, where the adversary is only given access to a noisy internal state. The latter interpretation of the d-probing model is much more realistic in that noise is an essential component of hardware implementations.

From the perspective of this work, there are also several reasons why the d-probing model seems a natural choice when trying to model glitch propagation. As we will show in Sections 3 and 4, our analysis is very much focused on relating the switching activity of single gates to the input variables that have caused it in the first place. According to how a gate changes its output, which takes into account glitches too, we will say that some variables are considered to be leaked, under certain circumstances. Thus, it is natural to think of such variables as being learned by an adversary who probes the output of that gate, in the spirit of the d-probing model.

3 Simulation of Signal Propagation

The choice of transients as a formalisation of signals relies on the operations that it is possible to define among them. Since the circuits we study are only formed of logic gates, we want those operations to preserve gates' functionalities. Therefore, we aim at building a function $\hat{f} : T^n \to T$ associated to a Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ whose inputs are n transients, namely $\underline{t} = (t_1, \ldots, t_n) \in T^n$.

Example 1 Let us suppose that two signals s_1 and s_2 are given as input to a gate implementing a Boolean function $f : \mathbb{Z}_2^2 \to \mathbb{Z}_2$. Firstly, they are fixed at constant values $b_1 \in \mathbb{Z}_2$ and $b_2 \in \mathbb{Z}_2$, respectively. Suddenly, s_1 changes from b_1 to $c \in \mathbb{Z}_2$, with $c \neq b_1$. This is represented by the transient $\mathbf{s}_1 = b_1 c$ which can be either 01 or 10. Then, the idea behind the function $\hat{f}$ is to emulate the behaviour of the function f, but taking as inputs the two transients $\mathbf{s}_1 = b_1 c$ and $\mathbf{s}_2 = b_2$ (seen as a length-one transient) and producing a transient with the highest number of switches, i.e. as if the highest number of glitches occurred. Note that we write a variable in boldface if it is seen as a transient and that bit concatenation is denoted by simply writing one bit after the other.

In the present work, we simply assume that the functionality discussed in Example 1 can be achieved. The idea is that, given two input transients $t_1 = a_1 \ldots a_{d_1}$ and $t_2 = b_1 \ldots b_{d_2}$, the first bit the gate computes is $f(a_1, b_1)$. This will be also called the initial stable state. Then the two inputs change to a_2 and b_2, respectively, and we have the freedom to decide which is the first one to affect the gate such that another change in the output (if any) is triggered. For a rigorous definition of the above, we refer the reader to our original paper [4] where all the steps are presented and proven.

Theorem 1 *Let $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ be a Boolean function. There always exists a function $\hat{f} : T^n \to T$ such that the output transient models the maximum number of switches that a gate implementing f might show. Moreover, $\hat{f}$ is well defined for any given input $\underline{t} = (t_1, \ldots, t_n) \in T^n$.*

3.1 Glitch-Counting Algorithm

The glitch-counting algorithm simulates the propagation of signals inside a circuit in terms of transients. First of all, a change in one or more inputs is assumed and represented as a transient. The glitch-counting algorithm assigns a transient to each gate as soon as the change reaches it. If the gate implements a Boolean function f, then the result is computed according to $\hat{f}$.

Given a circuit with m inputs and k gates, we denote by $\underline{X} = (X_1, \ldots, X_m)$ the vector of input variables and by $\underline{s} = (s_1, \ldots, s_k)$ the vector of state variables, which are the gates' outputs. We use boldface to distinguish when variables are used as transients, as in Example 1. Initially, suppose that the input $\underline{X}$ assumes the value $\underline{X} = \underline{a}' = (a'_1, \ldots, a'_m) \in \mathbb{Z}_2^m$, and that the state has the value $\underline{s} = \underline{b} = (b_1, \ldots, b_k) \in \mathbb{Z}_2^k$. We assume that the input changes to $\underline{a} = (a_1, \ldots, a_m) \in \mathbb{Z}_2^m$. We call this a *transition* and we denote it by $a'_1 \ldots a'_m \to a_1 \ldots a_m$. The goal is to study how glitches might propagate as a consequence of such a change.

The glitch-counting algorithm starts with the circuit in the initial stable state $(\underline{a}', \underline{b})$. The left-hand side is then set to the transient $\underline{\mathbf{a}} = (a'_1 a_1, \ldots, a'_m a_m)$ (note that in case $a'_i = a_i$ for some $i \leq m$, the transient obtained by concatenating them is only one bit) and is kept constant at that value for the duration of the algorithm. This stores how the input has changed. The right-hand side, instead, is stored in a vector of transients $\underline{\mathbf{s}}$ and is constantly updated throughout the duration of the algorithm: its value is modified in each position $\mathbf{s}_j$ according to the function $\hat{f}$, where f is the functionality of the gate computing s_j. We refer the reader

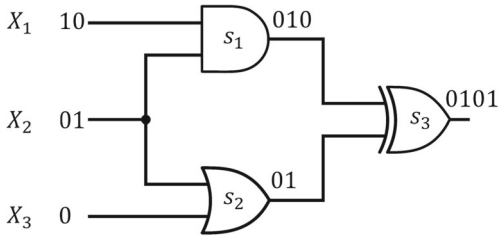

Fig. 1 Example of a glitch-counting algorithm's execution

to our original paper [4] and to the work of Brzozowski and Ésik [7] for the pseudo-code of the algorithm and further details. In this work, we opt for describing the algorithm by means of an example.

Example 2 Suppose that, in the circuit depicted by Fig. 1, the input changes from $\underline{a}' = (1, 0, 0)$ to $\underline{a} = (0, 1, 0)$; hence, the transition $100 \rightarrow 010$ occurs. The execution of the algorithm is summarised in Table 1, where each row represents one iteration of the cycle and each column refers to one variable (both input and state) of the circuit. The last two rows are identical, which is the termination condition of the algorithm. At each step, the algorithm computes the functions $\hat{f}$ of each gate for which previous transients are known. It follows the behaviour of real-world signal propagation; hence, earlier gates (i.e. closer to circuit inputs) are affected first. Indeed, the first row just represents the initial state (when only inputs have changed), the second one depicts a change in the first line of gates while in the third row, signals propagate till the last XOR. Figure 1 is a graphical representation of the final situation, which is the output of the algorithm without intermediate steps. Note that the final logic situation can be retrieved from Table 1 by extracting the last bit of each state variable.

We conclude the present section with a theorem stating the asymptotic running time of the glitch-counting algorithm. The proof is extensively discussed by Brzozowski and Ésik [7] and is then omitted here.

Theorem 2 (**Section 8 of** [7]) *Given a feedback-free circuit and a transition of its inputs, the glitch-counting algorithm always terminates. Moreover, it runs in $O(m + k^2)$ time where m is the number of inputs and k the number of gates.*

Table 1 Example of a glitch-counting algorithm's execution

h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$
0	10	01	0	0	0	0
1	10	01	0	010	01	0
2	10	01	0	010	01	0101
3	10	01	0	010	01	0101

4 LP Model

The glitch-counting algorithm was developed in the first place to prevent unnecessary power consumption by discarding netlists being particularly exposed to glitch propagation [7]. Our main contribution is the *LP (leakage path) model*, which is a mathematical abstraction that expands the functionalities of the glitch-counting algorithm and relates its simulations to the notion of leakage. Our result leads to a tool that allows to evaluate if a circuit has a critical leakage from the security point of view. The remaining of this section explains the structure of the LP model, which is formed of the following mathematical entities:

Input variables can trigger a signal propagation. If no input variable changes, no signal propagates and no power is consumed, therefore no leakage exists according to our power model.

Literals are sets of input variables. For each gate reached by a signal's change, a literal contains which variables have caused the change and could then be leaked.

Literifiers are the link between transients and leakage. Essentially, they relate the input and output transients of a gate to the appropriate literal.

The general idea behind the above three objects is the following. The process begins with a change in the input variables, which generates a signal propagation inside the circuit and affects some gates. The gates are then supposed to produce a new output based on the new inputs and their final result depends on which variables have changed and how. In this framework, literifiers are responsible to retrieve the variables involved and represent them via literals. Finally, Section 4.2 develops an argument according to which the above concepts are applied to a whole circuit, and not just to a single gate, so to relate them to the d-probing model.

4.1 Structure of LP Model

We now describe in detail each part of the LP model with respect to a single gate. This means that when we talk of input variables, we mean the variables that are directly given as inputs to it. The next subsection will prove a broader view, showing how to apply notions for single gates to a whole circuit. Following the same notation as the input variables of a circuit, we denote such variables by X_j and by $\mathbf{X}_j$ if they are seen as transients; we assume that $f : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_2$ is the Boolean function implemented by the gate and we

denote by $\hat{f} : T^n \to T$ the corresponding function among transients.

As stated in the introduction of this section, input variables are of great importance for both the glitch-counting algorithm, since nothing could be simulated without a change of theirs, and the LP model. In essence, they are the objects our study targets as we aim at following their propagation along the circuit.

Definition 2 Given a gate with n inputs, namely $X_1, \ldots, X_n$, we call *literal* any subset of $\{1, \ldots, n\}$. The set of literals is denoted by $I = \mathcal{P}(\{1, \ldots, n\})$.

Literals are finite sets of input variables. In a sense, they are the result we are looking for: the analysis of a circuit by means of the LP model consists in assigning a literal to each gate. Their utility stems from the fact that they list which input variables are responsible for the power consumption and could then be leaked according to our power model. This is strictly connected with the rationale behind transients. In both cases, we assume the worst possible scenario: transients are supposed to switch as if the worst possible combination of glitches occurred in the same way as literals list all variables being leaked in the worst possible case. It is clear from the above discussion that the core of the LP model is the way we assign literals to gates.

Literifiers are functions establishing which input variables are leaked by a gate, i.e. the ones having caused a change in its output. They depend on how the gate's inputs change, i.e. which transients enter in it, and on the implemented logic. First of all, we represent the input of a gate as the following vector of couples:

$$((t_1, l_1), \ldots, (t_n, l_n)) \in (T \times I)^n.$$

We call it *transient-variable representation*: the first component of each couple is a transient modelling how that input signal changes, while the second one is a literal listing the input variables responsible for that change.

Example 3 Recalling Fig. 1, the gate computing $\mathbf{s}_1 = 010$ has the following input according to the transient-variable representation.

$$((10, \{1\}), (01, \{2\}))$$

In Example 3, we have assumed that the literal of a circuit's input is just the singleton containing its index. As for now, the transient-variable representation is directly possible only for gates at height 1, i.e. whose inputs are inputs of the circuit itself. In that case, each literal is simply the singleton of a variable. In the next subsection, we will show a procedure similar to the glitch-counting algorithm to meaningfully apply literifiers also to gates whose inputs have already been processed. Such gates are said to have height grater than 1. Informally speaking, the height of a gate is inductively defined to be 1 if all its inputs are circuit inputs, and to be the maximum height of its inputs plus one otherwise. We intentionally omit any further formalisation to avoid heavy notations. As an example, in the circuit in Fig. 1, the AND and OR gates are at height 1 and the XOR is at height 2.

We refer the reader to our original paper [4] for a detailed and general description of how literifiers can be built for an arbitrary Boolean function among transients $\hat{f} : T^n \to T$. For the sake of simplicity, we limit the following discussion to the specific case of the gates AND, NOT, OR and XOR since a compact definition exists.

Definition 3 The literifier associated to a gate implementing the Boolean function $\texttt{AND} : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is defined as:

$$L_{\texttt{AND}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } t_j = 0 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid \ell(t_j) > 1\}$.

Intuitively, the upper branch in Definition 3 states that if there exists one input which is the fixed 0, then the output will be the fixed 0 no matter how other inputs change. Since the output is fixed, no power is consumed and the set of leaked variables is empty. Otherwise, the union of all literals corresponding to non-constant transients is returned. Since we are in the second branch, there is no constant 0 transient, which results in the rule excluding only literals being equal to the constant 1, as they do not contribute to the switching activity of an AND gate.

Example 4 Following Example 2, let us compute the literifier $L_{\texttt{AND}}((10, \{1\}), (01, \{2\}))$ associated to the gate computing s_1. A straightforward application of Definition 3 yields

$$L_{\texttt{AND}}((10, \{1\}), (01, \{2\})) = \{2\} \cup \{1\} = \{1, 2\}.$$

For the OR gate the argument is perfectly dual to the AND gate's, and then the literifier associated to it follows.

Definition 4 The literifier associated to a gate implementing the Boolean function $\texttt{OR} : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is defined as:

$$L_{\texttt{OR}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } t_j = 1 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid \ell(t_j) > 1\}$.

The NOT gate is clearly the easiest: if the input transient does not switch, so does the output, and then no power is consumed. Otherwise, the only possible literal is returned.

Definition 5 The literifier associated to a gate implementing the Boolean function $\texttt{NOT} : \mathbb{Z}_2 \to \mathbb{Z}_2$ is defined as:

$$L_{\texttt{NOT}}(t, l) = \begin{cases} \emptyset & \text{if } \ell(t) = 1 \\ l & \text{otherwise} \end{cases}$$

Finally, the XOR is slightly different than the AND and OR, since such a gate switches whenever at least one input switches. This restricts the cases in which $L_{\texttt{XOR}}$ returns the empty set.

Definition 6 The literifier associated to a gate implementing the Boolean function $\texttt{XOR} : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is defined as:

$$L_{\texttt{XOR}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \forall j \leq n, \ell(t_j) = 1 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid t_j \neq 0\}$.

4.2 Application to Circuits

We conclude this section by showing how to apply the LP model to a given circuit with m inputs and k gates. For instance in Fig. 1, on one hand, it is immediate that the transient-variable representation of gate computing s_1 is the one shown in Example 3, but on the other, it is less clear what it should be for gates whose inputs are not the inputs of the circuit, e.g. for the one computing s_3. We recall that we denote by $\underline{X} = (X_1, \ldots, X_m)$ the input variables and by $\underline{s} = (s_1, \ldots, s_k)$ the state variables of a circuit.

The idea is simply proceeding by height: the only gates we can directly compute literifiers for are those at height 1, since the input literals are just singletons of input variables. Once all literifiers at height 1 have been computed, we can apply those at height 2: their input literals can be either singleton of input variables or outputs of gates at height 1. This procedure always terminates as there are finitely many gates and is well-defined as there are no feedbacks.

Example 5 We conclude what Example 4 has begun by computing all literifiers of Example 2. The only other gate at height 1 is the one computing s_2, for which we have the following.

$$L_{\texttt{OR}}((01, \{2\}), (0, \{3\})) = \{2\}$$

We now have all the information to compute the literifier for the last gate.

$$L_{\texttt{XOR}}((010, \{1, 2\}), (01, \{2\})) = \{1, 2\} \cup \{2\} = \{1, 2\}$$

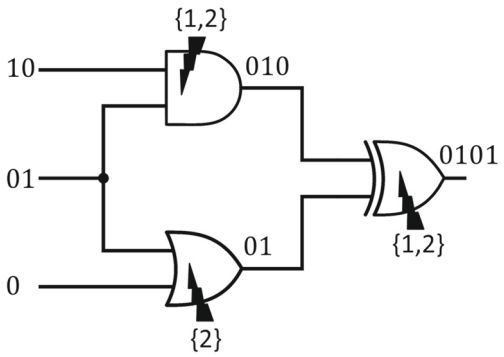

Fig. 2 Application of literifiers to a circuit

Figure 2 depicts the final outcome of the LP model applied to the circuit in Fig. 1 during transition $100 \to 010$. Essentially, the LP model adds one literal per gate to the output of the glitch-counting algorithm. They describe which input variables cause a particular gate to switch and whose values could then be leaked through the power consumption. Collecting such an information for all transitions gives the designer a powerful tool to predict possible flaws. In the next section, we deepen this discussion while providing a real-world use case. Note that there is a straightforward interpretation of literal in terms of the d-probing model: they are the set of variables (i.e. their value) that an adversary can learn by placing the probes on the output of the gate which produced them.

Final Remarks In the present subsection, we have shown how to practically apply the LP model to the netlist of a circuit. Although the example we have considered was trivial, the LP model is a formal tool to analyse netlists with an arbitrary number of inputs and gates in the d-probing model, where an ad hoc analysis would require much more effort. Once a netlist and an input transition are fixed, the LP model provides a list of variables based on which a risk assessment in the context of side-channel analysis is facilitated. As the next section will suggest, a full analysis would require the LP model to run over every non-trivial input transition, hence $2^{2m} - 2^m$ times where m is the number of inputs and where we have subtracted transitions from an input to itself as they clearly do not produce any consumption in our power model. Such exponential requirement is a drawback of our approach: a deeper insight will be given in Section 6. Finally, for a fixed transition, the overall complexity is asymptotically bounded by the running time of the glitch-counting algorithm, described in Theorem 2.

5 Case of Study: KECCAK

The present section provides an application of the LP model to KECCAK. We show, thanks to our tool, how an

unprotected implementation of KECCAK's non-linear layer is proved to be weak against side-channel attacks, and how glitches might also compromise security in the masked scheme. The reason why we chose to adopt KECCAK as our case of study mainly relies on it being deployed in real-world applications while still having a not too complex structure. It is then the ideal candidate for being a test bench.

KECCAK is a family of sponge functions that uses a permutation from a set of seven possible ones as a building block [3]. The permutations are defined over a state $s \in \mathbb{Z}_2^b$ where $b = 25 \times 2^\ell$ is called *width* of the permutation and $\ell \in \{0, \ldots, 6\}$. Each round is formed of five maps: three linear maps aiming at diffusion and dispersion, one non-linear map aiming at confusion and one addition with round constants. When it comes to implement sharing schemes, linear maps can be directly applied to each share separately. By contrast, non-linear maps need to handle every share to preserve correctness. Therefore, we focus on the only non-linear map of KECCAK, namely $\chi : \mathbb{Z}_2^5 \to \mathbb{Z}_2^5$ acting on groups of five bits of the state called *rows*. For a complete description of KECCAK, we invite the reader to refer to the work of Bertoni et al. [3].

The map χ can be seen as the parallel application of five identical maps each defined on three consecutive bits (modulo 5) of a row. Formally:

$$\chi_i : r_i \leftarrow r_i \oplus \bar{r}_{i+1} r_{i+2} \tag{1}$$

where $r \in \mathbb{Z}_2^5$ denotes a row of the KECCAK state and the index i is computed modulo 5. The bits r_i are called *native values*. For our analysis, it is important to note that the five instances of the map $\chi_i : \mathbb{Z}_2^3 \to \mathbb{Z}_2$ are completely independent; they do not share gates in their computation. As a result, we can focus on a specific χ_i without loss of generality.

5.1 Unshared χ_i

The first case that we study is the unshared χ_i, i.e. the Boolean function in Eq. 1. As both the glitch-counting algorithm and the LP model work with netlists, the first step in the analysis of Eq. 1 is to produce one. Assuming the naming convention at the beginning of Section 3.1, input vector $X = (X_1, X_2, X_3)$ corresponds to (r_i, r_{i+1}, r_{i+2}), while state vector $s = (s_1, s_2, s_3)$ corresponds to (NOT, AND, XOR). See Fig. 3 for a graphical representation of the latter.

It is trivial to see that any first order leakage at gate level is a leakage of a sensitive variable, then a critical leakage. Since χ_i has three input bits, the number of non-trivial transitions is $2^6 - 2^3 = 56$. We analyse two of them as example, namely $100 \to 011$ and $110 \to 111$. The execution of the glitch-counting algorithm for transition $100 \to 011$ is reported in Table 2 (left).

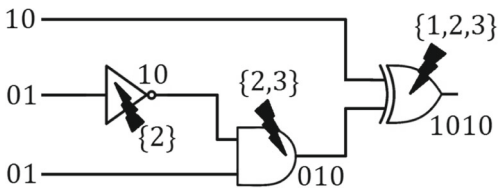

Fig. 3 χ_i circuit after LP model, transition $100 \to 011$

The LP model is then used: at first, literifier corresponding to s_1 is applied, since it is the only gate at height 1:

$$L_{\texttt{NOT}}(01, \{2\}) = \{2\}$$

Moving further to the gates at height more than 1, we compute $L_{\texttt{AND}}$ for s_2 and $L_{\texttt{XOR}}$ for s_3.

$$L_{\texttt{AND}}((10, \{2\}), (01, \{3\})) = \{2\} \cup \{3\} = \{2, 3\}$$
$$L_{\texttt{XOR}}((10, \{1\}), (010, \{2, 3\})) = \{1\} \cup \{2, 3\} = \{1, 2, 3\}$$

In Fig. 3, execution of both the glitch-counting algorithm and the LP model is depicted, in the case of the transition $100 \to 011$. Instead, in the case in which the inputs transition is $110 \to 111$, the glitch-counting algorithm shows that no glitch happens, as summarised in Table 2 (right).

The above two examples are meant to show two very different situations: in the first one, in Table 2 (left), it is evident how as soon as any gate switches; the subsequent power consumption will leak a sensitive variable. Indeed, an adversary could learn one by simply placing one probe (hence for $d = 1$) anywhere in the circuit. Table 2 (right), instead, interestingly shows how not all changes in inputs trigger some power consumption, although being restricted to very few corner cases. Similarly to the latter scenario, there are other three non-trivial input transitions that imply no leakage in the considered circuit, i.e. $111 \to 110$, $011 \to 010$ and $010 \to 011$. Hence, summarising, there are $56 - 4 = 52$ transitions that have some first order leakage, namely roughly the 81% of all transitions.

Although being no more than an exercise, the above discussions stress that the glitch-counting algorithm together with the LP model can be really fine grained in their analysis: they are able to precisely state the entity of the leakage even in the unprotected case which, surprisingly, does not happen as a result of every possible input transition. Nevertheless, there is a high possibility of such a critical leakage in this case, which is the reason why threshold implementations are implemented as countermeasures on χ_i [18].

5.2 χ with Two Shares

The first sharing scheme we adopt in our analysis is a two-share Boolean scheme, i.e. each row is split in two shares

Table 2 Glitch-counting algorithm's execution for the χ_i circuit, when the transition is $100 \rightarrow 011$ (left) and $110 \rightarrow 111$ (right)

h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$	h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$
0	10	01	01	1	0	1	0	1	1	01	0	0	1
1	10	01	01	10	01	10	1	1	1	01	0	0	1
2	10	01	01	10	010	101							
3	10	01	01	10	010	1010							
4	10	01	01	10	010	1010							

$a, b \in \mathbb{Z}_2^5$ such that $r = a \oplus b$ [2]. Our results can be easily generalised to many shares. In this setting, Eq. 1 can be masked as follows:

$$\begin{aligned} a_i &\leftarrow a_i \oplus \overline{a}_{i+1} a_{i+2} \oplus a_{i+1} b_{i+2} \\ b_i &\leftarrow b_i \oplus \overline{b}_{i+1} b_{i+2} \oplus b_{i+1} a_{i+2} \end{aligned} \tag{2}$$

where a straightforward computation shows that (2) are correct as Eq. 1 is simply retrieved by XORing them. If the order of operations was kept fixed from left to right then the above sharing scheme would be secure in the first order. However, if Eq. 2 were implemented in hardware, such condition could not be guaranteed, for instance because of glitches. This results in possible vulnerabilities when the values a_{i+2} and b_{i+2} are involved in the computation of the three-input XOR at the same time.

It can be easily seen from Eq. 2 that the two equations are symmetric; hence, the two netlists are identical. We will refer to them as being two branches of the implementation of Eq. 2. This also implies that we can focus only on the first branch without loss of generality, i.e. the one computing a_i. Similarly to what discussed for the unshared χ_i, the input vector $X = (X_1, X_2, X_3, X_4)$ corresponds to $(a_i, b_{i+2}, a_{i+1}, a_{i+2})$, while the components of the state vector $s = (s_1, s_2, s_3, s_4)$ correspond, respectively, to the NOT, upper AND, lower AND and XOR. See Fig. 4 for a graphical representation of the latter.

First of all an input transition is fixed among all the $2^8 - 2^4 = 240$ non-trivial possible ones. Then, the glitch-counting algorithm is applied as shown in Section 3.1 and all the transients are computed, one per gate. Table 3 reports the execution of the glitch-counting algorithm for the input transition $0110 \rightarrow 0001$.

At this point, suitable literifiers can be applied as described in Section 4.2, hence starting from gates at height 1. In our example, this means computing the literifiers corresponding to s_1 and s_2 first, respectively an AND and NOT literifiers.

$$\begin{aligned} L_{\texttt{AND}}((10, \{2\}), (10, \{3\})) &= \{2\} \cup \{3\} = \{2, 3\} \\ L_{\texttt{NOT}}(10, \{3\}) &= \{3\} \end{aligned}$$

There are two gates at height higher than 1: first we compute $L_{\texttt{AND}}$ for the gate computing s_3 and finally $L_{\texttt{XOR}}$ is applied.

$$\begin{aligned} L_{\texttt{AND}}((01, \{3\}), (01, \{4\})) &= \{3\} \cup \{4\} = \{3, 4\} \\ L_{\texttt{XOR}}((0, \{1\}), (10, \{2, 3\}), (01, \{3, 4\})) &= \{2, 3\} \cup \{3, 4\} = \\ &= \{2, 3, 4\} \end{aligned}$$

Figure 4 summarises the execution of both the glitch-counting algorithm and of the LP model for the transition $0110 \rightarrow 0001$.

To take the most out of the proposed method, a vulnerability definition based on critical combinations of variables needs to be formulated. This is checked among all the literals produced by the model, which has been run over all possible non-trivial input transition. Notice, however, that the sharing scheme outlined above is secure in the one-probing model if the order of operations is enforced, because at each step of the algorithm, each internal variable is independent of any sensitive ones. Unfortunately, glitches falsify such an argument.

A vulnerability of the circuit in Fig. 4 arises when the two variables a_{i+2} and b_{i+2} are processed in the same moment by the last XOR gate, as this could leak the value $a_{i+2} \oplus b_{i+2} = r_{i+2}$ which is unshared. As mentioned above, this would not be possible without glitches: they make an

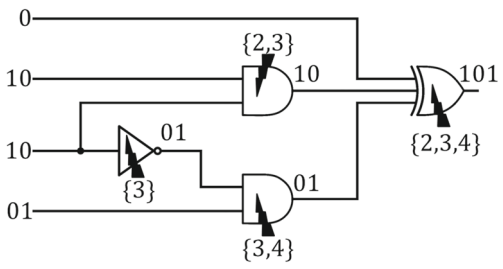

Fig. 4 One branch of two-shared χ_i circuit after LP model

Table 3 Glitch-counting algorithm's execution for the shared χ_i circuit

h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_4$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$	$\mathbf{s}_4$
0	0	10	10	01	1	0	0	1
1	0	10	10	01	10	01	0	1
2	0	10	10	01	10	01	01	10
3	0	10	10	01	10	01	01	101
4	0	10	10	01	10	01	01	101

attack feasible with a single probe at the output of the XOR. In our model, this translates to the existence of {2} and {4} in the same literal corresponding to the XOR gate, since X_2 and X_4 are the input variables corresponding to a_{i+2} and b_{i+2}. By running the model for all the $2^8 - 2^4$ non-trivial possible input transitions, we have found that 32 out of 240 match our vulnerability definition and could then lead to a critical first order leakage. At this point, the designer possesses valuable information to base security improvements on. In particular, leaving our gate-level abstraction, the designer can carefully tune place-and-route paths in order to minimise the occurrence and impact of those critical transitions. If such an operation is not feasible, the designer still has a valid and sound criterion why to switch to a higher number of shares (three in the case of KECCAK, since χ has degree 2).

The sharing scheme we have analysed [2] has not gained much popularity due to its weakness in the presence of glitches. However, our analysis is able to capture more details: we can quantify and list all those transitions threatening the security of unshared values. In this case, a designer could just patch them while being sure that all the others will never show a critical leakage of the first order even in the presence of glitches. We note that the possibilities for such a patch already exist in the literature. For instance, the work by Leiserson et al. [12] presented masked gates resilient to glitch propagation: it could be the case that a clever combination of our approaches might lead to beneficial results, for example by masking only those gates being critical under a certain vulnerability definition and not others, so to spare resources. Another possible heuristic approach would be to link the results shown by our model to practical considerations on actual vulnerability. This means that it might be possible to bound the SNR and other attacks' success metrics given the simulation provided by our tool, to infer on practical (in)feasibility of attacks. However, since our aim was just to exemplify the potentiality of our model, we consider the latter modifications as being out of scope for the present work, but an interesting future direction towards sound and lightweight countermeasures.

5.3 χ with Three Shares

To overcome the presence of glitches causing a leakage of the first order, generally the three-share Boolean scheme is adopted. Each KECCAK row r is split in three shares $a, b, c \in \mathbb{Z}_2^5$, such that $r = a \oplus b \oplus c$ [2]. Now Eq. 1 can be masked in the following way:

$$\begin{aligned} a_i &\longleftarrow b_i + \overline{b}_{i+1}b_{i+2} + b_{i+1}c_{i+2} + c_{i+1}b_{i+2} \\ b_i &\longleftarrow c_i + \overline{c}_{i+1}c_{i+2} + c_{i+1}a_{i+2} + a_{i+1}c_{i+2} \\ c_i &\longleftarrow a_i + \overline{a}_{i+1}a_{i+2} + a_{i+1}b_{i+2} + b_{i+1}a_{i+2} \end{aligned} \tag{3}$$

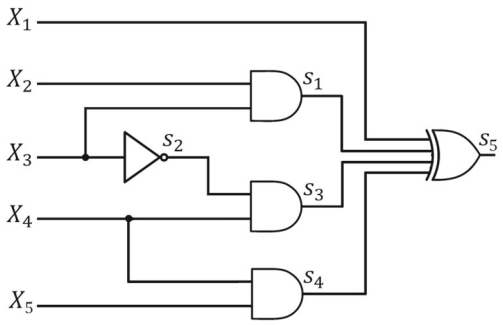

Fig. 5 Netlist of χ_i for one share, in the case of three shares

The XOR of equations in Eq. 3 allows to retrieve the χ_i function in Eq. 1. Moreover, each equation in Eq. 3 never processes all the shares of a native value. For example, the map producing a_i operates on one share of the native variable r_i (b_i), two shares of r_{i+1} (b_{i+1} and c_{i+1}) and two of r_{i+2} (b_{i+2} and c_{i+2}). Note that the missing share is exactly the one being output, coherently with the definition of threshold implementations [18, 19]. Since all the branches are symmetric, their netlists are the same. The one producing a_i is depicted in Fig. 5, where the input vector $X = (X_1, X_2, X_3, X_4, X_5)$ refers to $(b_i, c_{i+2}, b_{i+1}, b_{i+2}, c_{i+1})$.

Considering only one share function, no leakage of a native value can appear, since its shares are never involved in the computation of the four-input XOR by construction. As we have just mentioned, the three branches are symmetric; hence, we can focus our analysis on the two branches that produce a_i and b_i without loss of generality. If we monitor such circuits in at least two different points, it should be possible to observe some leakages that can give information on a native variable if combined together. In the literature, this is usually referred to as high-order leakage. Firstly, we jointly consider the two circuits, i.e. such that the inputs vector is $X = (X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8)$, that corresponds to the native variables vector $(b_i, c_{i+2}, b_{i+1}, b_{i+2}, c_{i+1}, c_i, a_{i+2}, a_{i+1})$.

Since the number of inputs is 8, the number of transitions is 2^{16}, and among them $2^{16} - 2^8 = 65280$ are non-trivial. As an example, we choose to show the behaviour of our tool on the transition 00111111 $\rightarrow$ 01101000: the execution of the glitch-counting algorithm is reported in Table 4, where we have adopted some simplifications to make the table smaller. The variables $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_4, \mathbf{X}_5, \mathbf{X}_6, \mathbf{X}_7$ and $\mathbf{X}_8$ are set, respectively, to 0, 01, 1, 10, 1, 10, 10 and 10; hence they are not reported.

The next step is to apply the literifiers, starting from gates at height 1, i.e. gates producing $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_4$ for the first circuit (equations on the left) and gates producing $\mathbf{s}_6, \mathbf{s}_7, \mathbf{s}_9$ for the second one (equations on the right).

Then, literals for gates at height more than one are computed too, i.e. for gates producing $\mathbf{s}_3, \mathbf{s}_5$ for the first

Table 4 Glitch-counting algorithm's execution for the two branches of the three-share χ_i

h	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$	$\mathbf{s}_4$	$\mathbf{s}_5$	$\mathbf{s}_6$	$\mathbf{s}_7$	$\mathbf{s}_8$	$\mathbf{s}_9$	$\mathbf{s}_{10}$
0	0	0	0	1	1	1	0	0	0	0
1	01	0	0	10	1	10	0	0	010	01
2	01	0	0	10	101	10	0	0	010	01010
3	01	0	0	10	101	10	0	0	010	01010

$L_{\text{AND}}((01, \{2\}), (1, \{3\})) = \{2\}$ $L_{\text{AND}}((10, \{7\}), (1, \{5\})) = \{7\}$
$L_{\text{NOT}}(1, \{3\}) = \emptyset$ $L_{\text{NOT}}(1, \{5\}) = \emptyset$
$L_{\text{AND}}((10, \{4\}), (1, \{5\})) = \{4\}$ $L_{\text{AND}}((01, \{2\}), (10, \{8\})) = \{2, 8\}$

circuit (upper equations) and $\mathbf{s}_8$, $s\mathbf{s}_{10}$ for the other (lower equations).

$$L_{\text{AND}}((0, \emptyset), (10, \{4\})) = \emptyset$$
$$L_{\text{XOR}}((0, \{1\}), (01, \{2\}), (0, \emptyset), (10, \{4\})) = \{2, 4\}$$

$$L_{\text{AND}}((0, \emptyset), (01, \{2\})) = \emptyset$$
$$L_{\text{XOR}}((10, \{6\}), (10, \{7\}), (0, \emptyset), (010, \{2, 8\})) = \{2, 6, 7, 8\}$$

Figure 6 summarises the execution of both the glitch-counting algorithm and the LP model for the transitions described in the example.

The first circuit can have a leakage of input variables X_2 and X_4 that refer to shares c_{i+2} and b_{i+2}, while in the second circuit, there can be a leakage of X_2, X_6, X_7 and X_8 corresponding to shares c_{i+2}, c_i, a_{i+2} and a_{i+1}. Then, with just two probes, an adversary could learn information on shares a_{i+2}, b_{i+2} and c_{i+2}, making possible to recover the native value r_{i+2}. Obviously, this is no longer a first order leakage, since an attacker has to use at least two probes to implement the attack.

Considering only two branches of the whole sharing scheme, there are some transitions that match the above vulnerability definition and could then lead to a critical high-order leakage of the two native values r_{i+1} and r_{i+2}. In particular, there are 6016 transitions producing a leakage on r_{i+1}, namely roughly 9.18% of all transitions, while there are 6144 transitions producing a leakage about r_{i+2}, roughly the 9.38%. Finally, 1024 of these transitions lead to leakage on both r_{i+1} and r_{i+2}, i.e. 1.56% of al l transitions.

In conclusion, analysing the three-sharing scheme for χ_i with the LP model, we have deduced that no critical first order leakage can happen, since by placing only one probe, it is not possible to recover information about all the shares of a native value. Instead, studying the propagation of glitches in two circuits, we have noticed that there can be a critical high-order leakage of some native variable. In particular, an adversary being able to place two probes can retrieve a variable which is correlated to a sensitive one, giving rise to an attack of the second order. This is a theoretical result shown by our model; hence, practical experiments to verify the existence of the above would be a very valuable future direction, also considering that the previous best attack against the three-sharing scheme is a third order attack by Bertoni et al. [1]. Furthermore, we notice that all the above discussions remain true even if any two branches are chosen to run the LP model on, not just the ones returning a_i and b_i. The latter fact reveals how such an attack can work equally well against all the three couples of chosen branches.

6 Computational Effort and Multi-output Circuits

Since our aim is not to find a specific method for KECCAK but a rather generic methodology, there are two further topics that need to be addressed: the computational complexity for a generic circuit and the applicability of the method to multi-output combinatorial circuits.

The former topic has been partially addressed in Section 3.1 for the glitch-counting algorithm (Theorem 2) and in Section 4.2 for the LP model. If we refer to KECCAK as a practical example and we think at an implementation performing one round in one clock cycle, the target combinatorial circuit is the concatenation of θ, one of the linear maps, and χ [3]. This combinatorial circuit can be seen as a circuit with 33 input bits and 1 output bit in the unprotected version, while the protected version using two shares is a 44-input circuit [2]. As described in Section 4.2, this would turn in computing the propagation of glitches through k gates for each of the $2^{2m} - 2^m$ non-trivial input transitions. Considering that the computation can be parallelised and the evaluation of the glitch-counting algorithm is not a very complex computation, we claim that the method could be applicable for a circuit with 44 inputs but would require a well-optimised implementation.

Multi-output circuits are also a very interesting target. In such circuits, there are gates contributing to the computation of different output bits. One approach for tackling these circuits is to divide the circuit in N independent circuits with single output, where N is the number of outputs of the

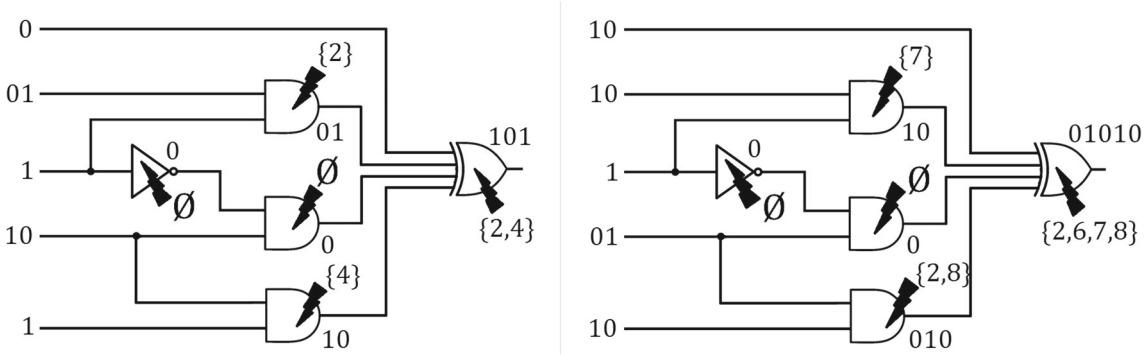

Fig. 6 Branches of the three-shared χ_i circuit after LP model; on the left, there is the first circuit producing a_i and on the right, the second one producing b_i

initial combinatorial logic, nothing prevents the model to be applied as it is to each of those separately, but meaningful vulnerability definitions would be required to correctly interpret the results.

7 Conclusions

In their work, Brzozowski and Ésik [7] have developed a mathematical structure to estimate the potential waste of power of a circuit due to glitches. Our first contribution is the expansion of such framework to include a formal definition of leakage. We have then defined a formal procedure to analyse circuits in the d-probing model which takes into account the effect of glitches on the order of operations. Our work analyses only the combinatorial logic and hence achieves a good level of generality since it is not affected by real-world constraints. As a consequence, the LP model allows to retrieve how much a given protection scheme can be weakened by glitches, thus enabling a deep analysis. Using the proposed methodology, a designer might explore alternative workflows for solving local problems of glitches instead of adopting more costly solutions.

Funding The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR). Furthermore, Marco Martinoli has been supported in part by the Marie Skłodowska-Curie ITN ECRYPT-NET (Project Reference 643161).

References

1. Bertoni G, Daemen J, Debande N, Le T, Peeters M, Assche GV (2012) Power analysis of hardware implementations protected with secret sharing. In: IEEE/ACM International symposium on microarchitecture, MICRO, pp 9–16
2. Bertoni G, Daemen J, Peeters M, Assche GV (2010) Building power analysis resistant implementations of Keccak. In: Second SHA-3 candidate conference
3. Bertoni G, Daemen J, Peeters M, Assche GV (2013) Keccak. In: Johansson T, Nguyen PQ (eds) EUROCRYPT 2013, LNCS, vol 7881. Springer, Heidelberg, pp 313–314, https://doi.org/10.1007/978-3-642-38348-9_19
4. Bertoni G, Martinoli M (2016) A methodology for the characterisation of leakages in combinatorial logic. In: Carlet C, Hasan MA, Saraswat V (eds) Proceedings of the 6th International Conference on Security, Privacy, and Applied Cryptography Engineering, SPACE 2016, Hyderabad, India, December 14–18, 2016. Springer International Publishing, Cham, pp 363–382. ISBN: 978-3-319-49445-6, https://doi.org/10.1007/978-3-319-49445-6_21
5. Bilgin B, Gierlichs B, Nikova S, Nikov V, Rijmen V (2014) Higher-order threshold implementations. In: Sarkar P, Iwata T (eds) ASIACRYPT 2014, Part II, LNCS, vol 8874. Springer, Heidelberg, pp 326–343, https://doi.org/10.1007/978-3-662-45608-8_18
6. Bilgin B, Gierlichs B, Nikova S, Nikov V, Rijmen V (2014) A more efficient AES threshold implementation. In: Pointcheval D, Vergnaud D (eds) AFRICACRYPT 14, LNCS, vol 8469. Springer, Heidelberg, pp 267–284, https://doi.org/10.1007/978-3-319-06734-6_17
7. Brzozowski J, Ésik Z. (2003) Hazard algebras. Formal Methods Syst Des 23(3):223–256
8. Duc A, Dziembowski S, Faust S (2014) Unifying leakage models: from probing attacks to noisy leakage. In: Nguyen PQ, Oswald E (eds) EUROCRYPT 2014, LNCS, vol 8441. Springer, Heidelberg, pp 423–440. https://doi.org/10.1007/978-3-642-55220-5_24
9. Ishai Y, Sahai A, Wagner D (2003) Private circuits: securing hardware against probing attacks. In: Boneh D (ed) CRYPTO 2003, LNCS, vol 2729. Springer, Heidelberg, pp 463–481
10. Kocher PC, Jaffe J, Jun B (1999) Differential power analysis. In: Wiener MJ (ed) CRYPTO'99, LNCS, vol 1666. Springer, Heidelberg, pp 388–397
11. Kocher PC, Jaffe J, Jun B, Rohatgi P (2011) Introduction to differential power analysis. J Cryptograph Eng 1(1):5–27
12. Leiserson AJ, Marson ME, Wachs MA (2014) Gate-level masking under a path-based leakage metric. In: Batina L, Robshaw M (eds)

CHES 2014, LNCS, vol 8731. Springer, Heidelberg, pp 580–597, https://doi.org/10.1007/978-3-662-44709-3_32
13. Mangard S, Oswald E, Popp T (2008) Power analysis attacks: tevealing the secrets of smart cards, vol 31. Springer Science & Business Media
14. Mangard S, Popp T, Gammel BM (2005) Side-channel leakage of masked CMOS gates. In: Menezes A (ed) CT-RSA 2005, LNCS, vol 3376. Springer, Heidelberg, pp 351–365
15. Mangard S, Pramstaller N, Oswald E (2005) Successfully attacking masked AES hardware implementations. In: Rao JR, Sunar B (eds) CHES 2005, LNCS, vol 3659. Springer, Heidelberg, pp 157–171
16. Mangard S, Schramm K (2006) Pinpointing the side-channel leakage of masked AES hardware implementations. In: Goubin L, Matsui M (eds) CHES 2006, LNCS, vol 4249. Springer, Heidelberg, pp 76–90
17. Moradi A, Poschmann A, Ling S, Paar C, Wang H (2011) Pushing the limits: a very compact and a threshold implementation of AES. In: Paterson KG (ed) EUROCRYPT 2011, LNCS, vol 6632. Springer, Heidelberg, pp 69–88
18. Nikova S, Rechberger C, Rijmen V (2006) Threshold implementations against side-channel attacks and glitches. In: Ning P, Qing S, Li N (eds) ICICS 06, LNCS, vol 4307. Springer, Heidelberg, pp 529–545
19. Nikova S, Rijmen V, Schläffer M (2009) Secure hardware implementation of non-linear functions in the presence of glitches. In: Lee PJ, Cheon JH (eds) ICISC 08, LNCS, vol 5461. Springer, Heidelberg, pp 218–234
20. Prouff E, Roche T (2011) Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In: Preneel B, Takagi T (eds) CHES 2011, LNCS, vol 6917. Springer, Heidelberg, pp 63–78
21. Rabaey JM, Chandrakasan AP, Nikolic B (2002) Digital integrated circuits, vol 2. Prentice Hall, Englewood Cliffs
22. Reparaz O (2016) Detecting flawed masking schemes with leakage detection tests. In: Peyrin T (ed) FSE 2016, LNCS, vol 9783. Springer, Heidelberg, pp 204–222, https://doi.org/10.1007/978-3-662-52993-5_11
23. Reparaz O, Bilgin B, Nikova S, Gierlichs B, Verbauwhede I (2015) Consolidating masking schemes. In: Gennaro R, Robshaw MJB (eds) CRYPTO 2015, Part I, LNCS, vol 9215. Springer, Heidelberg, pp 764–783. https://doi.org/10.1007/978-3-662-47989-6_37
24. Tiwari M, Wassel HM, Mazloom B, Mysore S, Chong FT, Sherwood T (2009) Complete information flow tracking from the gates up. In: Proceedings of the 14th international conference on architectural support for programming languages and operating systems, pp 109–120